Automatic Correction to Misspelled Names: A 4'th Generation Language Approach

Michael Allen Bickel *

The MITRE Corporation

Summary

An information theoretic likeness measure is defined as an inner product on a data space created from a table of valid names. Using this measure, a 4GL procedure searches this data base space for the nearest correctly spelled name.

Introduction

A large data base application which uses personal names as part of a key must ensure that these names are spelled consistently. Otherwise, retrieval becomes unnecessarily complicated, since most computers interpret a misspelled name to mean a different person. An algorithm for automatically identifying two different spellings of the same name may seem to be impossible and unrealistic; however, this new approach is more than 95% successful and is the subject of this article.

* 1120 Nasa Road 1
MITRE 6'th floor
Houston, Texas 77558

(NASA-CR-182322) AUTOMATIC CORRECTION TO MISSPELLED NAMES: A 4TH GENERATION LANGUAGE APPROACH (Mitre Corp.) 15 p

N88-70137

Unclas
00/82 0114215

With an existing computer network, NASA is creating an on-line data base to catalog and track the multitude of tasks necessary for each shuttle launch. The straightforward way to index these tasks is to use the task assignee's name as the primary key. This results in fast retrieval for each assignee, but it also forces tasks assigned to "Jones, Robert A." to be stored separately from tasks assigned to "Jones, Bob A.". (Name is used throughout thisarticle as a data type with format "lastname, firstname middleinitial.")

One solution to this problem, caused by nicknames and misspellings, is a table of valid names, with the requirement that each assignee entered into the data base be a member of this names table; any attempt to assign a task to a person not in this table will fail. A single data entry point has the responsibility for entering new employee's names into this table before any tasks can be assigned to them. This names table and the membership requirement has been specified in a schema declaration using NOMAD2, a 4'th generation data base language and management system in use at the Johnson Space Center. This language can optionally branch to user defined procedures in the event of an error or a membership failure. The membership failure branch in this application reads the user's screen for the misspelled name which failed the membership test and then searches for the nearest valid

name in the names table. If the search is successful, the procedure permits the user to correct the name on the screen and then the membership test is passed; otherwise, a subset of the names table is displayed for a user selection.

A major consideration in designing a names table search algorithm is its execution time. The names table at NASA contains almost one thousand names and yet the response time must be less than one or two seconds. The algorithm presented here assumes that the first letter of lastname is known with certainty and therefore the search is limited to this section of the names table. Even with this reduction, sometimes a hundred names must be searched.

To achieve acceptable response times, it is critical that only the functions supplied with the data base language be employed; these functions are optimized by the 4GL compiler. This application requires the repeated computation of an inner product of two vectors, one of which contains only zeroes and ones. This type of product can be calculated very rapidly, without multiplication, by using two numerical array (vector) functions that are included in NOMAD2: array summation (term by term addition of two arrays) and array sum (scalar sum of the elements in an array). The skillful use of the NOMAD2 data type NAV (data not available), enables this

inner product to be replaced by sum and summation. This data type is functionally appropriate since all calculations involving NAV arguments result in a NAV answer and all numerical NAV items may be replaced by zero using the single command 'ZNAV'. This 4GL procedure is logically equivalent to bit by bit 'anding' of two 27 bit integers and then using the result to mask a weight array before summing. The geometric basis for using an inner product is discussed in the remarks at the end of this article.

Heuristic

Some misspellings, such as the omission of a character, will shift correctly spelled characters into incorrect positions. In these cases, the position of any character within the name is less valuable as search information than the actual character. This is because a position by position comparison usually disagrees on characters to the right of the omission: for example, if "Addison, Bob G." is the intended name and "Adison, Bob G." is the entered name, then a position by position comparison agrees only on the left two characters.

In addition, a less frequently occurring character, such as a "y", is more valuable as search information than a more frequently occurring character, such as an "e". Therefore,

each character is assigned a number from 3 to 9 as an information weight; 3 is the weight for "a,e,i,n,o,s,t" ; 4 is the weight for "d,h,l,r,u" ; 5 is the weight for "c,f,g,m,p,w" ; 6 is the weight for "b,v" ; 7 is the weight for "k,q" ; 8 is the weight for "j,x,y" ; 9 is the weight for "z"; and 0 is the weight for all non-alpha characters.

Based upon these weights, this algorithm computes a likeness value for any two names as follows: every likeness is initially set to zero; then, for each character in the alphabet, if that character is present in both names, the weight for that character is added to the likeness value. Using this method, and pairing the misspelled name with each name in the reduced names table, a likeness value is computed for each pair. From the pair with the largest likeness value, the name most probably intended by the user is displayed and the user is queried whether this is the correct name.

Notice that each character of the alphabet contributes to the likeness value at most once, regardless of its multiplicity in either name. Also, the first character in lastname entered on the screen always contributes since each lastname remaining in the reduced names table begins with this character.

Before ending this heuristic, two points essential to the success of this method should be mentioned explicitly. First, nobody has a name which contains every letter of the alphabet; if such a name existed, it would always produce the maximum LIKENESS regardless of the misspelled name. Second, the number of names in the names table is small compared to the total number of names possible. This implies a sparseness in the names table that translates into a sparseness in the name space; i.e., most names in the name space are well separated from each other. These are the reasons that the nearest valid name in the names table is usually the name the user intended.

Pseudo-code for NAMESTABLE

(procedure for entering new employee names)

1) Fetch the new employee NAME entered on the screen
2) Call HASH (NAME , MASK) to compute the array MASK
   Let MASK = MASK * WEIGHT
   (where * is array product,i.e. term by term multiply)
3) Insert NAME and MASK in the NAMES TABLE file
   Return

- HASH (NAME , MASK)
  Initialize the array MASK(1 to 27)=0
  For each character in NAME, set MASK (character)=1
  where character denotes the position of this character in

the alphabet (i.e., 1 to 26 for letters A through Z) and 27 is assigned for any number or other symbol (all non-alpha characters)

Return

Pseudo-code for NAMESEARCH

(procedure executed on membership failure)

1) Fetch the NAME entered on screen (lastname, firstname mi.)
   Reduce the NAMES TABLE assuming that the first character of lastname is correct
2) Call HASH (NAME , MASK) to compute the array MASK
   Change the zeroes in MASK to NAVs
   Then change the ones in MASK to zeroes
   Save as MASKSAVE
3) For each NAME in the reduced NAMES TABLE
   Fetch the accompanying MASK
   Call the scalar function LIKENESS (MASKSAVE , MASK)
   Find the pair with the largest LIKENESS
   (Special case to handle tied LIKENESSes)
4) Display the pair of names having the largest LIKENESS
   Query the user whether this is the intended name
   If correct, move the intended name to the screen
   Else display the reduced NAMES TABLE for a search
   Return

- **LIKENESS (MASKSAVE , MASK)** **!Returns a scalar!**

```
TEMP = MASKSAVE + MASK                    !term by term add!
TEMP = ZNAV(TEMP)                  !change NAV's to zeros!
LIKENESS = SUM(TEMP)               !sum all array elements!
Return
```

Notice how the NAV entries in MASKSAVE mask the unwanted weights in MASK and also how the zero entries in MASKSAVE pass the wanted weights in MASK. Changing the NAV entries in TEMP to zero and summing, results in precisely the sum of the unmasked weights in MASK.

Remarks

The weight assigned to a character is based upon the relative frequency with which that character occurs. These relative frequencies can be computed from the names table or defaulted to typical English frequency distributions. The weights calculated from the frequencies in the names table at Johnson Space Center are remarkably close to the weights computed from standard English frequencies, except for 'J' and 'Q'. First order information theory [1] calculates the weight of a character according to the following formula:

$$WEIGHT(character) = - LOG(frequency(character))$$

where LOG is a base 2 logarithm.

Different names do not always produce different MASKs; for example, "Anders, Dan R." and "Andersen, Sean D." both have the same MASK. Hence, this method may not always find the intended name even when extra values based upon positional agreements are added to LIKENESS. Significant positions for checking agreements include: the first character of each firstname; the middle initials; the second character of each lastname; and the third character of each lastname. Using these refinements, this algorithm is generally more successful (i.e., fewer collisions) than the SOUNDEX algorithm which uses a hash that is only four characters long [2].

Using valid names from the names table, three different simulations are used to generate misspelled names for timing and accuracy measurements. SIM1 generates a misspelled name by inserting a random letter at a random position (anywhere except the first position). SIM2 generates a misspelled name by omiting one letter at a random position (anywhere except the first position). SIM3 generates a misspelled name by exchanging the letters at any two random positions (anywhere except the first position). From each name in the names table, a misspelled name is generated and then this misspelling is used for searching the names table. Counters tabulate the number of successful searches and the time

elapsed. The time measured is the total response time (not simply the CPU time) and includes program overhead, system and network processing,and database I/O. These times are dependent, not only upon the comlexity of the algorithm, but also upon the number of applications and the number of users on the host during these timing measurements. Hence, a good and a worst response time were computed during moderate and heavy loads. During light usage, the reponse time is considerably better than the minimum time reported. The following table summarizes these measurements:

	SIM1	SIM2	SIM3
times	0.7 sec	1.1 sec	0.8 sec
times	1.5 sec	1.9 sec	1.4 sec
% error	4.5 %	3.0 %	2.4 %

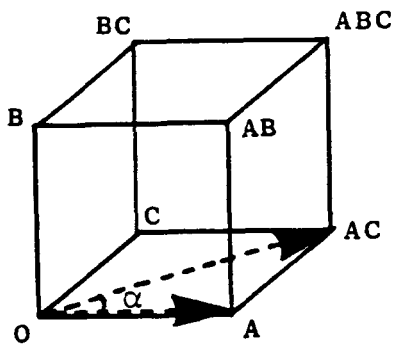
The effectiveness of this algorithm is due to the geometrical meaning of inner product within the data space of names. Via the subroutine HASH, any name (not necessarily a name in the names table) may be mapped to a corner of a 26 dimensional cube called the name space as follows: A corner of this cube is chosen as the origin for the name space and

each of the 26 dimensions (i.e., 26 different directions from this origin) is associated with one character of the alphabet. Then, by using the MASK associated with each name as a set of coordinates, each name is mapped to a point (i.e., a corner) in the name space. Thus, each name is associated with a point which determines a vector in the name space beginning at the origin and ending at that point. A pair of names determines two vectors which lie in a plane containing the origin and these two vectors meet at the origin in a definite angle.

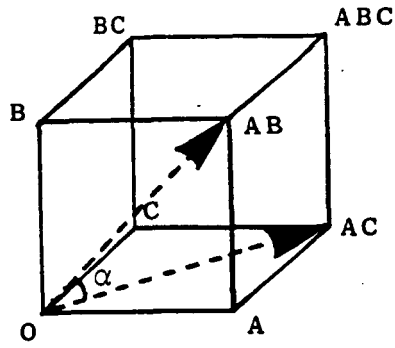
The inner product of two vectors is equal to the product of their lengths times the cosine of the angle between these two vectors. If a correct spelling exists, it must be found within a small region of the name space surrounding the misspelled name entered on the screen (the assumption is that the nearest name is probably the intended name). In any small region of the name space, the lengths of the name vectors are almost constant, hence the inner product varies as the cosine of the angle between the name vectors. The nearest name in the names table corresponds to the name vector subtending the smallest angle and this maximizes the cosine factor in the inner product. Thus, the nearest name always corresponds to the maximum inner product and is located by finding the largest LIKENESS.

A three dimensional example may be used to graphically illustrate this idea. Suppose that the alphabet consists of only three letters {A,B,C} and that they each have unit weights. Suppose also, that "AC" is the misspelled name entered on the screen and that the names table contains the following names: "A", "BA", "AB", and "ABC". After reducing the names table, it is necessary to calculate LIKENESSes for three pairs of names: ("AC","A"), ("AC","AB"), and ("AC","ABC"). These LIKENESSes and their associated angles are graphed in the following diagrams.

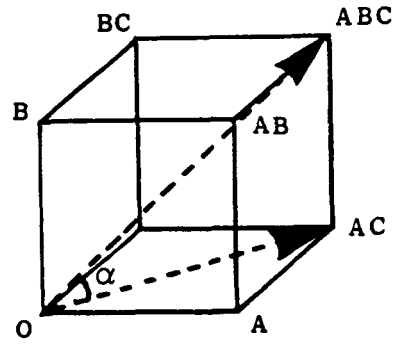
(AC,A)



(AC,AB)



(AC,ABC)



MASKSAVE(AC) +MASK(A) =TEMP	MASKSAVE(AC) +MASK(AB) =TEMP	MASKSAVE(AC) +MASK(ABC) =TEMP
AC ==> 0 NAV 0	AC ==> 0 NAV 0	AC ==> 0 NAV 0
A ==>+1 +0 +0	AB ==>+1 +1 +0	ABC ==>+1 +1 +1
TEMP = 1 NAV 0	1 NAV 0	1 NAV 1
ZNAV (TEMP)==>1 0 0	1 0 0	1 0 1
SUM (TEMP)= LIKENESS= 1 +0 +0=1	1 +0 +0=1	1 +0 +1=2
ANGLE= 45	60	35

The pseudo-code equations are repeated here for convenience:

```
TEMP = MASKSAVE + MASK                !term by term add!
TEMP = ZNAV(TEMP)                   !change NAV's to zeros!
LIKENESS = SUM(TEMP)               !sum all array elements!
```

Thus, ABC is probably the name intended, since it yields the largest LIKENESS (i.e., it subtends the smallest angle).

A name containing all of the letters in the alphabet, like this example, would always produce the largest LIKENESS; however, in reality, no valid name contains all of the characters in the alphabet. Put another way, names tend to be mapped to corners in the name space that are close to the origin and names become very scarse near the corner opposite the origin (this corner corresponds to the name containing all of the letters). This fact and the sparseness of names in the name space enable this method to be successful.

A quote from Niklaus Wirth's new book is completely appropriate for ending this article.[3] During the analysis of key transformations, Wirth says "... considerable confidence in the correctness of the laws of probability theory is needed by anyone using the hash technique." Thus, even though this method may seem to magically find the correct spelling, there is no magic in algorithms, and its success is due to mathematics.

Acknowledgments

I wish to thank all of my colleagues at MITRE who contributed to this article through their discussions. Special thanks to Walter Bays, Roger Pearson, Frank Venezia, George Stark, and Walt Colquitt. Also, I wish to thank my NASA liason, Elena Hufstetler, who always beleived that it

should be possible. This work was done at MITRE and supported by Air Force contract F 19628-86-C-0001 NASA T-3589M.

References

[1] Young, John F., "Information Theory",Wiley Interscience, New York (1971) pp. 44-45.

[2] Knuth, Donald E., "Art of Computer Programming", Addison Wesley, Reading Massachusetts (1973) vol.3 pp. 391-392.

[3] Wirth, Niklaus. "Algorithms and Data Structures", Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1986) pp. 277.